

Using Parameterized Signatures to Express Modular Structure

Mark P. Jones

Department of Computer Science, University of Nottingham,
University Park, Nottingham NG7 2RD, England.

mpj@cs.nott.ac.uk

<http://www.cs.nott.ac.uk/Department/Staff/mpj/>

Abstract

Module systems are a powerful, practical tool for managing the complexity of large software systems. Previous attempts to formulate a type-theoretic foundation for modular programming have been based on existential, dependent, or manifest types. These approaches can be distinguished by their use of different quantifiers to package the operations that a module exports together with appropriate implementation types. In each case, the underlying type theory is simple and elegant, but significant and sometimes complex extensions are needed to account for features that are important in practical systems, such as separate compilation and propagation of type information between modules.

This paper presents a simple type-theoretic framework for modular programming using parameterized signatures. The use of quantifiers is treated as a necessary, but independent concern. Using familiar concepts of polymorphism, the resulting module system is easy to understand and admits true separate compilation. It is also very powerful, supporting high-order, polymorphic, and first-class modules without further extension.

1 Introduction

Large scale software development obtains significant benefits from the ability to break programs into collections of modules, each of which can be designed, implemented and understood as individual, often reusable, components. Formal studies of the theoretical foundations for modular programming provide valuable insights into the design of more powerful, and more effective module systems for practical programming languages.

During the past decade, there have been several attempts to provide type-theoretic foundations for modular programming [23, 16, 21, 7, 22, 27, 1, 6, 13, 15, 3]. The main proposals can be distinguished by their use of different constructs to describe the type of a module. To illustrate these alternatives, suppose that we wish to construct a complex number

package that provides a collection of operations:

```
type Complex c
= { mkCart, mkPolar :: Float → Float → c;
    re, im          :: c → Float;
    mag, phase     :: c → Float; ... }
```

There are three established methods for packaging a collection of operations like this together with a suitable implementation type:

- **Existential types (opaque, or weak sums)** [23, 5]. In a module of type $\exists c. \text{Complex } c$, the existential quantifier conceals the identity of the implementation type c by making it *abstract*. This approach allows packages to be treated as first-class values and is fully compatible with separate compilation. However, for the purposes of modular programming, existentials are often *too* good at hiding implementation types because they do not allow adequate propagation of type information between packages [16].
- **Dependent types (transparent, or strong sums)** [18, 16, 21]. A module of type $\Sigma c. \text{Complex } c$ is represented by a pair $\langle \tau, M \rangle$ containing the type τ used to represent complex numbers and an implementation M of type $\text{Complex } \tau$ for the complex number operators. The ability to include type components in modules makes this approach very powerful — too powerful in fact to permit compile-time type checking without careful restrictions and extensions to ensure a suitable phase distinction [7] and to support features like sharing [14]. Even then, it is still not possible to support true separate compilation [2] or to use modules as first-class values [22, 12].
- **Manifest types (translucent sums)** [6, 13] are an attempt to bridge the gap between the weak and strong sum approaches described above. A module type $\exists c = \tau. \text{Complex } c$ is much like an existential, except that it exposes the fact that a particular implementation type τ was chosen. An extra rule in the type system can be used to make the implementation type abstract by coercing a manifest type to a standard existential.

We should also mention the treatment of modules in Standard ML (SML) which provides one of the most powerful module systems in widespread use. The overall design is often explained in terms of dependent types, but it is not easy

To appear in the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 21-24, 1996.

Copyright © 1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., Fax +1 (212) 869-0481, or (permissions@acm.org).

to discern their use in the formal definition [20]. Instead, the definition uses a semantics based on freshly generated tokens called stamps to account for the concepts of sharing and generativity. This approach works well as a method for the specification and implementation of a type checker, but seems too operational for other purposes. In addition, it has proved to be “remarkably difficult to modify or extend” [6].

The advantages and disadvantages of each of the three approaches described above will be described in more detail in the following section. The important point to note for now is that, in each case, the type of a module, package or structure is given by an expression of the form $Qt.f(t)$ for some parameterized signature $f(t)$ and some quantifier Q . Many of the complications that we have referred to above are caused by the fact that these quantifiers can be ‘overly protective’, limiting the ability to propagate type information between modules.

In this paper, we show how parameterized signatures such as *Complex* c can be used as the types of modules, treating quantifiers as a separate concern. The result is a simple, yet powerful type system for modular programming. Of course, we still need a mechanism like existential quantification to support the definition and use of abstract datatypes, but this can be dealt with in different ways and need not be so closely tied to individual modules. Polymorphism, represented by universal quantification, is another important part of the type system that allows us:

- To express sharing constraints and to describe the propagation of type information.
- To reflect the independence of the implementation of a parameterized module from the implementation of its parameters.
- To allow the definition of polymorphic modules, a useful feature that is not permitted in SML [11].

We believe that the use of polymorphism is natural and easy to understand, particularly for programmers who are already familiar with the type systems of the core languages of SML, Haskell, or similar languages.

Another important benefit of parameterized signatures is that they ensure a clear separation between static and dynamic semantics; the type system allows a programmer to associate a given module with a particular collection of types, but module values do not include type components. As a result, we do not have to worry about the problems of establishing a phase distinction, and true separate compilation is possible because the way that a module can be used is completely specified by its signature. This also means that modules can be treated as first-class values.

The remaining sections of this paper are as follows. We begin in Section 2 with a more detailed analysis of previous attempts to provide a type-theoretic basis for modular programming. This helps to understand the strengths and weaknesses of the different approaches and to clarify our own goals in the design of a module system. Section 3 describes how parameterized signatures fit into the picture and gives some examples to illustrate their use. A formal presentation of the type system is presented in Section 4. Concerns about the suitability of a type system using parameterized signatures as a basis for modular programming are addressed in Section 5. Finally, Section 6 concludes with pointers to future work.

2 Background and motivation

In our introductory comments, we have already mentioned several previous attempts to provide a type-theoretic foundation for modular programming. In this section, we take a more detailed look at each of these proposals, explaining how the design of each system has been motivated by the strengths and weaknesses of its ancestors. Our aim is to provide a survey of related work, and motivation for the use of parameterized signatures.

All of this work rests on fundamental assumptions about the nature and purpose of module systems. Certainly, it would be wrong to regard a particular type system as a basis for modular programming unless it provides some, if not all, of the following:

- A mechanism to support separate compilation and some form of namespace management.
- A mechanism to enable the decomposition of large programs into small, reusable units in a way that is resistant to small changes in the program.
- A mechanism for defining abstractions.

Throughout this paper, we argue that the use of parameterized signatures is consistent with such goals. Support for separate compilation is ensured by maintaining a clear separation between types and values, while effective program decomposition is supported by the use of higher-order and nested polymorphism. Powerful abstractions can be defined using parameterized structures. The system does not provide a built-in notion of abstract datatypes but, instead, allows this to be dealt with using other methods.

2.1 Existential types

One way to formalize the process of hiding the implementation of an abstract datatype is to use an existential type [23, 5]:

$$\mathbf{type} \text{ ComplexPkg} = \exists c. \text{Complex } c.$$

Informally, the existential typing indicates that there is a type c with operations of type *Complex* c defined on it. At the same time, it prohibits a programmer from making any assumptions about the implementation type. Formally, the properties of existentials are described by typing rules, based on standard logical rules for existential quantifiers. The following rule is often described as an introduction rule because it introduces a new occurrence of the \exists quantifier in the conclusion:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t.\tau)}$$

Note that the implementation type τ' for the abstract type is discarded and does not appear anywhere in the conclusion.

The requirement that N has a polymorphic type in the following elimination rule ensures that we do not make any assumptions about the now-hidden implementation type; N behaves uniformly for all choices of t :

$$\frac{\Gamma \vdash M : \exists t.\tau \quad \Gamma \vdash N : \forall t.\tau \rightarrow \tau' \quad t \notin TV(\tau')}{\Gamma \vdash \mathbf{open} \ M \ \mathbf{in} \ N : \tau'}$$

Existential types completely hide the identity of implementation types. For example, the types c and c' in the body of the following expression cannot be identified, even though they come from the same term cpx of type $ComplexPkg$:

```

open  $cpx$  in  $\Lambda c.\lambda x : Complex\ c.$ 
open  $cpx$  in  $\Lambda c'.\lambda y : Complex\ c'.$ 
...

```

To further emphasize this behaviour, suppose that we want to define a package of complex number arithmetic, constructed from an arbitrary implementation of complex numbers. The obvious way to describe this is to use a function:

```

 $compArith$       ::  $ComplexPkg \rightarrow ArithPkg$ 
 $compArith\ cpx$  = open  $cpx$  in  $\Lambda c.\lambda p : Complex\ c.$ 
                  {  $add = \dots$  }

```

where:

```

type  $ArithPkg$  =  $\exists a.Arith\ a$ 
type  $Arith\ a$  = {  $add$  ::  $a \rightarrow a \rightarrow a;$ 
                   $neg$   ::  $a \rightarrow a; \dots$  }.

```

Given an implementation cpx of type $ComplexPkg$, we can use the expression $compArith\ cpx$ to obtain a package for arithmetic on complex numbers. Unfortunately, this has no practical use because the typing rules for existentials make it impossible to construct any values to which the add and neg functions of the resulting package can be applied! The type system does not capture the equivalence of the type of complex numbers used in cpx and the type of values that can be manipulated by $compArith\ cpx$.

An alternative approach to existential typing, using *dot notation* in place of the **open** construct described above, has been investigated by Cardelli and Leroy [4]. The dot notation allows us to identify the implementation types of two packages if they have the ‘same name’. This avoids the first problem illustrated above, but not the second. Dot notation is also limited by the unavoidably conservative notions of ‘same name’ that are needed to ensure decidability of type checking, and is not very well-behaved under simple program transformations.

2.2 Dependent types

Motivated by problems with existential types, MacQueen [16] argued that dependent types provide a better basis for modular programming. In this framework, structures are represented by pairs $\langle \tau, M \rangle$ containing both a type component τ and a term M whose type may depend on the choice of τ . Structures of this form can be viewed as elements of a dependent sum type, described informally by:

$$\Sigma t.f(t) = \{ \langle \tau, M \rangle \mid M \text{ has type } f(\tau) \}.$$

The typing rules for dependent sums are standard (see [18], for example) and can be written in the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash \langle \tau', M \rangle : (\Sigma t.f(t))} \quad \frac{\Gamma \vdash M : (\Sigma t.f(t))}{\Gamma \vdash \mathit{snd}\ M : f(\mathit{fst}\ M)}$$

The introduction rule on the left is very similar to the corresponding rule for existentials except that the implementation type, τ' , is captured in the structure $\langle \tau', M \rangle$ in the

conclusion. The elimination rule on the right indicates that, if M is a structure of type $\Sigma t.f(t)$, then the second component, $\mathit{snd}\ M$, of M has type $f(\mathit{fst}\ M)$, where $\mathit{fst}\ M$ is the first component of M . Informally, dependent sums are more powerful than existentials because of this ability to name the type component $\mathit{fst}\ M$ of a structure M . However, this also means that the type component of a structure is no longer abstract.

In a sense, a simple treatment of modules using dependent types is actually *too* powerful for practical systems because it interferes with separate compilation. More precisely, it makes it more difficult to separate compile-time type-checking from run-time evaluation. To illustrate this, we recast the previous definitions of complex number and arithmetic types using dependent sums to obtain:

```

type  $ComplexPkg$  =  $\Sigma cpx.Complex\ cpx$ 
type  $ArithPkg$    =  $\Sigma a.Arith\ a$ 
 $compArith$       ::  $ComplexPkg \rightarrow ArithPkg$ 
 $compArith\ c$    =  $\langle \mathit{fst}\ c, \{ add = \dots \} \rangle$ 

```

The $compArith$ function is an example of a parameterized module, or a *functor* in the terminology of SML. At first glance, this definition suffers from the same problems as the previous version using existentials; the type $ComplexPkg \rightarrow ArithPkg$ does not reflect the fact that the type components of the argument and result structures are the same. However, this information can be obtained by carrying out a limited degree of evaluation during type checking. For example, if $c = \langle \tau, M \rangle$, then:

$$\mathit{fst}\ (compArith\ c) = \mathit{fst}\ \langle \tau, \{ add = \dots \} \rangle = \tau.$$

To ensure that static type checking is possible, it is important to distinguish compile-time evaluation of this kind from arbitrary run-time execution of a program. Unfortunately, treating a functor as a function of type $(\Sigma s.f(s)) \rightarrow (\Sigma t.g(t))$ does not reflect this separation; in general, a type of this form may include elements in which the type component of the result depends on the value component of the argument. As an alternative, Harper, Mitchell and Moggi [7, 24] have shown that a suitable *phase distinction* can be established by modelling functors from $\Sigma s.f(s)$ to $\Sigma t.g(t)$ as values of type $\Sigma h.(\forall s.f(s) \rightarrow g(h(s)))$ where h ranges over functions from types to types, and describes the compile-time part of the functor.

As the example above shows, it is sometimes necessary to inspect the implementation of a structure to find the value of a type component. Not surprisingly, this means that it is impossible to provide true separate compilation for SML [2]. Even the smartest recompilation scheme proposed by Shao and Appel [26] does not permit true separate compilation because it delays some type checking, and hence the detection of some type errors, to link-time.

The need for type sharing constraints in functor definitions is motivated by similar problems. Formal parameters cannot be inspected at compile-time because their actual values are not known until compile time. Instead, identities between type components must be written explicitly using sharing equations. Further extensions to the basic theory of dependent types are needed to deal with this and other ideas including generativity, polymorphism, abstraction, higher-order modules, and modules as first-class values.

2.3 Manifest types

Recent proposals for *translucent sums* by Harper and Lillibridge [6] and *manifest types* by Leroy [13] provide a compromise between existential typing and dependent sums, allowing the programmer to include additional type information in the signature for a structure. For these systems, we use an introduction rule of the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t = \tau'.\tau)}$$

Notice that, unlike the previous cases, the implementation type τ' appears in the inferred type although this can be hidden by coercing it to a standard existential type:

$$\frac{\Gamma \vdash M : (\exists t = \tau'.\tau)}{\Gamma \vdash M : (\exists t.\tau)}$$

Manifest types provide good support for abstraction and separate compilation, although the underlying theories seem quite complex. One of the main technical problems has been the difficulty in providing sufficiently general and accurate types to capture the *fully transparent* behaviour of higher-order functors that is predicted by operational frameworks [27, 17]. In recent work, Leroy has shown how a calculus of manifest types can be modified to avoid this problem [15]. His solution requires an extension of the type language to include functor application, further blurring the distinction between types and terms; that is, between static and dynamic semantics. This work, and an alternative solution, are discussed in more detail in the next section.

3 Parameterized Signatures

In each of the approaches described above, the type of a module is given by an expression of the form $Qt.f(t)$ where $f(t)$ is a signature, parameterized by t , and Q is a quantifier. The goal of this paper is to show that we can use parameterized signatures like $f(t)$ as building blocks for a module type system and treat quantifiers as a separate issue.

For practical reasons, it is common to group related functions together in a single structure mapping variables to values. It follows that a parameterized signature $f(t)$ will usually be a record type pairing variable names with appropriate type schemes. There are two important aspects of our approach that distinguish it from other attempts to use record types to explain modular structure (for example, the work of Aponte [1]):

- **Higher-order polymorphism:** In the general case, we will want to use both types and type constructors as signature parameters. This is easily dealt with using a kind system as will be discussed in Section 4. There is no additional burden on the programmer to supply explicit kind information, because this can be inferred automatically from a program text. We already have considerable experience with such systems from work with *constructor classes* [10] which uses the same ideas; we know that they work well in practice.
- **Nested polymorphism:** In the general case, we will want to be able to define structures with polymorphic components and to use these structures as first-class

values; for example, as function arguments. It is well known that standard techniques for type inference do not allow function arguments with polymorphic types. Fortunately, it is fairly easy to deal with this in our system, first, by allowing the programmer to supply explicit type information, and second, because the use of polymorphism is clearly signaled by the presence of record types. This subject is discussed in more detail in Section 4.4.

We can illustrate the use of both of these features with the following signature which provides a representation for monads [28]:

$$\begin{aligned} \text{type } Monad \ m & \\ = \{ \text{bind} & :: \forall a.\forall b.m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b; \\ & \text{unit} :: \forall a.a \rightarrow m \ a \} \end{aligned}$$

Note that the kind inference mechanisms referred to above can be used to infer that the parameter m of the *Monad* signature is a unary type constructor, while the universally quantified variables a and b in the type of *bind* represent arbitrary types.

3.1 Polymorphism and sharing

With the examples of previous sections in mind, we might use a structure of type *Complex* c as an implementation of complex numbers, and structures with types of the form *Arith* a to describe the implementation of arithmetic operations on values of type a . The *compArith* function discussed in some detail above can now be treated as a polymorphic function:

$$\begin{aligned} \text{compArith} & :: \forall t.Complex \ t \rightarrow Arith \ t \\ \text{compArith } cpx & = \dots \end{aligned}$$

Because the same variable, t , appears as a parameter to both the *Complex* and *Arith* signatures, it is clear that the type of values that the arithmetic operations in the result can be applied to is the same as the type of complex numbers that are provided as an argument; this achieves much the same result as a type sharing constraint in SML. In addition, the fact that t is universally quantified ensures that we do not make any assumptions about the implementation of complex numbers. Thus polymorphism is useful as a way of expressing the independence of the implementation of a module from the implementation of its imports.

Parameterized signatures can also be used to express information about the propagation of type information in ways that are not possible with sharing equations in SML. For example, in his original work on manifest types [13], Leroy comments on the difficulty of handling a higher-order functor:

$$\begin{aligned} \text{signature } S & = \text{sig type } t; \dots \text{end} \\ \text{functor } apply & (\text{functor } f(x:S):S \\ & \quad \text{structure } a:S):S \\ & = f \ a \end{aligned}$$

The problem here is finding a way to propagate information about the relationship between the t components of the argument and result structures of f . This example has been further addressed in recent work on full transparency and higher-order modules [3, 15]. Leroy's solution is to use a

manifest type to specify that the t component of a structure $apply\ f\ a$ is the same as the t component of $f\ a$; this requires an extension to allow the use of functor applications in type expressions.

There is actually another way to solve this problem, as can be seen using parameterized signatures:

```
signature S t = sig ... end
functor apply(functor f(x:S t):S u
              structure a:S t):S u
  = f a
```

Here, the t component from the original code is represented by a parameter of the signature S . The definition of the $apply$ functor is polymorphic in the variables t and u , and this captures the desired relationship between the types of the arguments and result of $apply$ in a direct and concise manner. As a further comment, notice that, because structures can now be used as first-class values, there is no need to make a distinction between functors and ordinary function definitions. In fact, we could have defined $apply$ as an ordinary function:

```
apply      :: ∀a.∀b.(a → b) → a → b
apply f x  = f x
```

If we have already defined structures i and b , of types $S\ Int$ and $S\ Bool$, respectively, then we can use either version of $apply$ to determine that $apply\ (\lambda x.x)\ i$ has type $S\ Int$, and that $apply\ (\lambda x.b)\ i$ has type $S\ Bool$.

Another advantage of a module system based on parameterized signatures is the ability to provide a simple treatment for the definition and use of modules with polymorphic types. Recent work by Kahrs [11] shows that simple forms of polymorphism that can be used in the core language of SML are not permitted in the module language. Kahrs gives examples to show why polymorphism at the module level is useful and shows how it can be supported by extending the language with a new, general construct for describing the binding position of type variables. (In fact, this same construct could also be used as another alternative to Leroy's recent proposals [15] to handle the $apply$ functor discussed above.) We can illustrate the basic idea by observing that the type component t in any SML structure matching the signature:

```
signature I
= sig
  type t
  id    :: t → t
end
```

must be fixed to some specific type, t . As a result, it is impossible to define a structure s that matches I and such that $s.id$ is the polymorphic identity function.

Fortunately, this problem does not occur in our framework and a direct translation of the example here (and of those in Kahrs' paper) yields the desired form of polymorphism without any further work. The corresponding parameterized signature is just:

```
type I t = { id :: t → t }
```

and we can define a structure:

```
s :: ∀t.I t
s = struct id = λx.x end
```

Not only does this structure have a polymorphic type — we can also use the value $s.id$ as a polymorphic identity function of type $\forall t.t \rightarrow t$. For reasons of space, we have been forced to restrict our attention to a very simple example; we refer the reader to Kahrs' paper [11] for more compelling applications of this form of polymorphism.

3.2 Relationship with manifest types

The use of parameterized signatures is closely related to the system of manifest types described in Section 2.3. To understand this comment, we should think of a manifest type of the form $\exists t = \tau.\tau'$ as a kind of 'local definition', much as if it had been written **let** $t = \tau$ **in** τ' , or as a convenient notation for the result of a substitution $[\tau/t]\tau'$. Now let us repeat the rule for \exists -introduction and the rule for coercing a manifest type to an existential, both of which were discussed in previous sections:

$$\frac{\Gamma \vdash M : (\exists t = \tau'.\tau)}{\Gamma \vdash M : (\exists t.\tau)} \quad \frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t.\tau)}$$

From our current perspective, these rules are the same! With this observation in mind, it might appear that we have nothing to gain by adding manifest types to a type system that already includes existentials. However, there are two reasons why such a claim could be considered as misleading:

- For the purposes of the underlying type theory, some form of quantification is necessary in the proposed calculi for manifest types and translucent sums to account for the inclusion of type components in module values. This is at odds with our approach, which does not allow modules to contain type components.
- One may argue that the syntax for manifest types is better suited to modular programming because it avoids the awkwardness of large numbers of parameters, and may require fewer changes to the source of a program if a particular module is changed, for example, by adding type components. However, as we describe in Section 5.1, exactly the same benefits can be obtained in a simple and elegant fashion with parameterized signatures by packaging groups of parameters into record-like structures.

Another advantage, in theory, of the manifest type notation is that a type like $\exists t = \tau.\tau'$ might be much more concise and readable than the expanded form $[\tau/t]\tau'$ if τ is a complex type expression or if t appears several times in τ' . Again, in practice, this is not a serious issue because the languages that we are interested in (for example, SML and Haskell) already include facilities for defining type abbreviations or synonyms, and these can be used to achieve the desired effect.

3.3 Simple Examples

So far, we have discussed the motivation and theoretical aspects of parameterized signatures without many examples to show how corresponding structures might be defined. Not surprisingly, there are several different notations that we might choose from. For the purposes of this paper, and to facilitate easy comparison, we will adopt a SML-like syntax. However, to emphasize the distinction between static

and dynamic semantics, we separate out type declarations and value definitions, in the style of Haskell. Also, because structures are first-class values and there is no distinction between the module and core languages, we can omit the (now redundant) **structure** and **functor** keywords.

It is useful to start with a comparison between local definitions (**let** bindings) and structures:

```

let decls      struct
in  expr      decls
end

```

The collection of declarations, *decls*, introduced in each of these expressions will be type-checked in exactly the same way. The only difference is that, in a **let** construct the declared values are used immediately in the body *expr*, while in a structure they are packaged up for later use.

First, here is an implementation of complex numbers using the standard Cartesian representation:

```

rectCpx  :: Complex (Float, Float)
rectCpx = struct
    mkCart x y = (x, y)
    mkPolar r θ = (r cos θ, r sin θ)
    re (x, y) = x
    ...
end

```

As it stands, the implementation type of *rectCpx* is captured explicitly in its type. Later, in Section 5.3, we will describe how to make this type abstract, either completely by using an existential, or partially by giving it a name without revealing how it is implemented.

The next example is a fragment of the definition of the *compArith* function, which can be used to construct a complex arithmetic package from an arbitrary complex number package:

```

compArith  :: Complex t → Arith t
compArith c
= struct
    plus z1 z2 = c.mkCart (c.re z1 + c.re z2)
                  (c.im z1 + c.im z2)
    ...
end

```

Finally, the following definition specifies the structure of the list monad:

```

listMonad  :: Monad List
listMonad
= struct
    unit x = [x]
    (x : xs) 'bind' f = f x ++ (xs 'bind' f)
    [] 'bind' f = []
end

```

All of the examples given here should seem straightforward—and, of course, that is just what we want!

4 Formal development

This section provides a brief formal description of the type system that is put forward in this paper as a basis for modular programming.

4.1 Kinds and constructors

To support higher-order polymorphism, we need to allow the use of variables in type expressions to represent, not just arbitrary types, but also type constructors. Following standard techniques, we use a system of *kinds*, κ , to distinguish between different forms of type constructor:

$$\begin{array}{l} \kappa ::= * \quad \text{the kind of all (mono)types} \\ \quad | \kappa_1 \rightarrow \kappa_2 \quad \text{function kinds} \end{array}$$

Intuitively, the kind $\kappa_1 \rightarrow \kappa_2$ represents constructors that take a constructor of kind κ_1 and return a constructor of kind κ_2 .

For each kind κ , we have a collection of constructors C^κ (including constructor variables α^κ) of kind κ given by:

$$\begin{array}{l} C^\kappa ::= \chi^\kappa \quad \text{constants} \\ \quad | \alpha^\kappa \quad \text{variables} \\ \quad | C^{\kappa' \rightarrow \kappa} C^{\kappa'} \quad \text{applications} \\ \quad | \{x_i :: \sigma_i\} \quad \text{signatures, } \kappa = * \end{array}$$

Other than requiring that the function space constructor \rightarrow be included as an element of $C^{* \rightarrow * \rightarrow *}$, we do not make any assumption about the constructor constants χ^κ in the grammar above.

The symbol σ ranges over the set of type schemes described by the grammar:

$$\begin{array}{l} \tau ::= C^* \quad \text{monotypes} \\ \sigma ::= \forall \alpha^\kappa. \sigma \quad \text{polymorphic types} \\ \quad | \tau \end{array}$$

Note that this corresponds very closely to the way that most type expressions are already written in Haskell. For example, *List a* is an application of the constructor constant *List* to the constructor variable *a*. In addition, each constructor constant has a corresponding kind. For example, writing (\rightarrow) for the function space constructor and $(,)$ for pairing we have:

$$\begin{array}{l} \text{Int, Float, ()} ::= * \\ \text{List} ::= * \rightarrow * \\ (\rightarrow), (,) ::= * \rightarrow * \rightarrow * \end{array}$$

The syntax for constructors also includes expressions of the form $\{x_i :: \sigma_i\}$ of kind $*$, which is intended as a convenient abbreviation for record types of the form $\{x_1 :: \sigma_1; \dots; x_n :: \sigma_n\}$. These will be used primarily to assign types to structure values. Note the use of type schemes rather than simple types; this allows the system to support structures with polymorphic components. We should also mention that it would be possible, in theory, to encode module types as tuples without introducing the extra syntax for records. However, labelled tuples are perhaps more convenient in practice, and we have chosen to reflect this directly in our formulation of the type system.

Type checker implementations usually include tests to ensure that type expressions are well-formed, for example, that a particular constructor is supplied with an appropriate number of arguments. In the current setting, this can be reformulated as the task of checking that a constructor expression has kind $*$. The apparent mismatch between the explicitly kinded constructor expressions specified above and the implicit kinding used in examples can be resolved by a process of kind inference; that is, by using standard techniques to infer kinds for user defined constructors without the need for programmer-supplied kind annotations. The same approach has been used with considerable success in both the theory and practice of constructor classes [10].

4.2 Terms

For the purposes of this paper, it is sufficient to restrict our attention to a simple λ -calculus, extended with two constructs, one for building structures, and another for selecting structure components:

$M ::=$	x	<i>variables</i>
	$M M$	<i>application</i>
	$\lambda x.M$	<i>abstraction</i>
	let $x = M$ in N	<i>local definition</i>
	$M.x$	<i>selection</i>
	struct $x_i = M_i$ end	<i>structures</i>

The index notation in the last line of this grammar is used to reflect the fact that a structure may have multiple components.

4.3 Typing rules

With the definitions of the previous sections in place, we can use standard notation to specify the typing rules of our system in Figure 1. Note the use of the symbols τ and σ to restrict the application of certain rules to types or type schemes, respectively. The condition that $\alpha^\kappa \notin CV(A)$ in

(var)	$\frac{(x:\sigma) \in A}{A \vdash x : \sigma}$
$(\rightarrow E)$	$\frac{A \vdash E : \tau' \rightarrow \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau}$
$(\rightarrow I)$	$\frac{A_x, x:\tau' \vdash E : \tau}{A \vdash \lambda x.E : \tau' \rightarrow \tau}$
(let)	$\frac{A \vdash E : \sigma \quad A_x, x:\sigma \vdash F : \tau}{A \vdash (\mathbf{let} \ x = E \ \mathbf{in} \ F) : \tau}$
$(\{\}E)$	$\frac{A \vdash E : \{x_i :: \sigma_i\}}{A \vdash E.x_j : \sigma_j}$
$(\{\}I)$	$\frac{A \vdash E_j : \sigma_j \quad \text{for all } j}{A \vdash \mathbf{struct} \ x_i = E_i \ \mathbf{end} : \{x_i :: \sigma_i\}}$
$(\forall E)$	$\frac{A \vdash E : \forall \alpha^\kappa. \sigma \quad C \in C^\kappa}{A \vdash E : [C/\alpha^\kappa]\sigma}$
$(\forall I)$	$\frac{A \vdash E : \sigma \quad \alpha^\kappa \notin CV(A)}{A \vdash E : \forall \alpha^\kappa. \sigma}$

Figure 1: Typing rules

rule $(\forall I)$ is necessary to avoid universal quantification over a variable that is constrained by the type assignment A ; the expression $CV(A)$ denotes the set of all constructor variables appearing free in A .

4.4 Type inference

Although it really has little to do with the design of a module system itself, some readers may be concerned about the effects of adding higher-order and nested polymorphism to the type system of a language that is based on the use of

type inference. In fact, the first of these, does not cause any difficulty at all because we have used a weak form of higher-order polymorphism that avoids the undecidability of higher-order unification. The second, nested polymorphism, requires some form of explicit type information (although nothing more than would be required in an appropriate extension of SML).

4.4.1 Use of explicit type information

The need for explicit type information in programs using records will already be familiar to SML programmers; the definition of SML requires that the shape of any record—a complete list of its fields—can be determined at compile-time. What we have described here is a natural generalization of this; we require not only the names of all of the fields, but also the types for every field that is referenced.

Type annotations are not necessary in many simple examples. For example, the following program type checks without any additional type information:

```
f x = struct
      h z = [z]
      u = x
    end
v y = m.h (m.h m.u)
      where m = f y
```

In this case, we can calculate the following types for the components in the structure value in the definition of f :

$$\begin{aligned} h &:: \forall t. t \rightarrow [t] \\ u &:: a \end{aligned}$$

where a is the type of the argument x . Thus:

$$f :: \forall a. a \rightarrow \{h :: \forall t. t \rightarrow [t]; u :: a\}$$

and it follows that v has type $\forall a. a \rightarrow [[a]]$.

In practice, explicit type annotations are only required for the definition of recursive or mutually recursive structures (which are not permitted by the SML module system) or for functions that manipulate structure values (corresponding to SML functor definitions where explicit type information is also required in SML, or to higher-order or first-class modules which are not supported by SML). For example, the type signature accompanying the following definition cannot be omitted:

```
makeUnit      :: a  $\rightarrow$  Monad m  $\rightarrow$  m a
makeUnit x u = u.unit x
```

On the other hand, we are free to store values of some type *Monad* m in data structures such as lists and to use many higher order functions, for example $\lambda z. \text{map} (\text{makeUnit } z)$, without further type annotations.

Some may question the need for explicit type information in a language that is based on a Hindley-Milner type system, but we do not believe that this will have any significant impact on programmers:

- Some form of explicit type information is already necessary in many languages based on Hindley-Milner typing. For example, this includes the overloading mechanisms of Haskell; the treatment of records, arithmetic, and structures in SML; and the notations used to define new datatypes in each language.

- Explicit type annotations are only required in situations where they would already be required by programs using the SML module system, or in programs that cannot be written with SML modules.
- Despite the fact that it is not necessary, the use of type annotations in implicitly typed languages like ML and Haskell is widely recognized as ‘good programming style’, and many programmers already routinely include type declarations in their source code. The type assigned to a value serves as a useful form of program documentation. In addition, this gives a simple way to check that the programmer-supplied type signatures, reflecting intentions about the way an object will be used, are consistent with the types obtained by type inference.

It is important to find a formal mechanism that can be used to describe when additional type information is required for a given program, and to indicate what form it should take. This can already be achieved by selecting a particular implementation of the type inference algorithm. However, this risks over-specification and we would prefer to find a more abstract, and less operational alternative.

5 Concerns about modularity

In addition to formal concerns, there are a number of pragmatic issues that must be addressed in the design of a module system. We claim that the system of parameterized structures presented here is suitable as a module system, but our current prototype is not sufficiently complete to have allowed us to obtain practical experience with it on a large scale programming project. The aim of this section is to do ‘the next best thing’ by discussing a number of issues that have been suggested as important properties for module systems, and showing how they are dealt with in the framework of this paper.

5.1 Signature parameters and sharing

It is easy to find applications of SML that use modules with fairly large numbers of type components; being forced to specify a value for every parameter of the corresponding signature in our framework would be awkward and inconvenient. The SML notation, and in particular sharing constraints, are also more robust in the sense that, if extra type components are added to a signature, then we do not necessarily have to modify the program as we might, for example, to add an extra parameter to each use of a parameterized signature.

In fact, if we package parameters together in records, then parameterized signatures can offer the same advantages. The only change that we need to make to the formal development in Section 4 is to add new syntax for records of constructors:

$$\begin{aligned} \kappa & ::= \dots \\ & \quad | \{t_i :: \kappa_i\} \quad \text{record kinds} \\ C^\kappa & ::= \dots \\ & \quad | C^{\{t_i :: \kappa_i\}}.t_j \quad \text{selection, if } \kappa = \kappa_j \\ & \quad | \{t_i = C^{\kappa_i}\} \quad \text{construction, } \kappa = \{t_i :: \kappa_i\} \end{aligned}$$

It is important to understand the difference between records of constructors $\{t_i = C^{\kappa_i}\}$ and signatures $\{x_i :: \sigma_i\}$, both of

which can appear in types; this is reflected by the fact that the two expressions will be assigned different kinds.

With this approach, sharing constraints can be understood as a form of qualified type [9]:

$$\text{prog} \quad :: \quad (r.x = s.y) \Rightarrow \text{SIG } r \rightarrow \text{SIG}' s$$

The constraint $(r.x = s.y)$ appearing here indicates that the x and y fields of r and s , respectively, must be equal. Note that we do not need to mention any other fields of the r and s records, or even to know that there are any other fields.

Another interesting observation is that the type of prog does not make any references to term language constructs. This would not be true in SML where the names of functor parameters serve an additional role as labels for the signatures appearing in a type. This also helps to explain the problems of typing the apply functor in Section 3.1 because there is no obvious label for the signature of the result of applying f to x in the definition:

$$\begin{aligned} \text{functor } \text{apply}(\text{functor } f(x:S):S \\ \text{structure } a:S):S \\ = f a \end{aligned}$$

With our approach, each signature has an obvious label; the record of constructors that are used as its parameters. As we have already seen, this makes it easy to assign a useful and general type to apply without any extensions to the language.

5.2 Type components

In the system that we have been describing in this paper, modules do not include type components. Instead, we use signature parameters to capture relationships between structure components and implementation types. This is a significant departure from some of the previous work, for example, in systems based on dependent types where type components play a central role. However, we argue that much can be accomplished *without* type components. This, we believe, is also more in the spirit of the Hindley-Milner type system. For example, in Milner’s original work [19], types are used as a purely semantic notion, representing subsets of a semantic domain, not as any concrete form of value.

A key observation is that type definitions within a module can be lifted to the top-level. For example, consider the following SML fragment:

```
structure s
= struct
  type T          = Int
  data List a    = Nil | Cons a (List a)
  ...
end
```

Despite appearances, the type synonym T and the type constructor $List$ are *not* local to the definition of s . At any point in the program where s is in scope, these type constructors can be accessed by the names $s.T$ and $s.List$, respectively. Renaming any references to these types and their constructors in the body of s , we can lift these definitions to the top-level, to obtain the following definitions:

```
type s.T          = Int
data s.List a    = s.Nil | s.Cons a (s.List a)
structure s      = ...
```

In effect, all that the datatype definitions in the original SML program accomplish is to define top-level datatypes in which the type and value constructor names are decorated with the name of the structure in which they are defined.

In some situations, renaming is not sufficient to allow type definitions to be lifted to the top-level. For example, the *List* datatype in the following functor definition involves a ‘free variable’, the type $x.T$, a component of the argument structure x :

```

functor  $f(x:SIG) : SIG'$ 
= struct
  data  $List = Nil \mid Cons\ x.T\ List$ 
  ...
end

```

The solution in this case is to add an extra parameter to the datatype definition before moving it to the top-level, as shown in the following code fragment. This is just a form of λ -lifting [8, 25]:

```

data  $f.List\ t = f.Nil$ 
            $\mid f.Cons\ t\ f.List$ 
 $f$ 
 $f\ x$ 
  ::  $SIG\ t \rightarrow SIG'\ (f.List\ t)$ 
  = ...

```

Notice how the parameterized signatures in the type for f capture the relationship between the types involved in the argument x and those involved in the result $f\ x$. Because the form of higher-order polymorphism described in Section 4.1 allows type constructors to be used as both signature and datatype parameters, the same technique can be used to deal with type constructor components of functor arguments.

In SML, the two functor definitions above are not equivalent; SML adopts a notion of generativity, producing a new type constructor each time the functor is applied to an argument. Thus two definitions:

```

structure  $s_1 = f(x)$ 
structure  $s_2 = f(x)$ 

```

will produce structures with incomparable type components. In truth, when we use an SML functor to ‘generate’ a new datatype, we are in fact constructing a new instance of a fixed datatype, which is then hidden, in essence, by a form of existential quantification. There is no way to express the *List* type produced by applying f to an appropriate argument structure in the notation of SML, so we are forced to package up instantiation of the actual implementation type, $f.List$, and hiding of the resulting type as a single operation.

Lifting type definitions to the top-level allows us to express the type components of the result of functor applications; for the example above, if x has type $SIG\ t$, then both s_1 and s_2 have type $SIG'\ (f.List\ t)$. We are then free to treat the question of whether we wish to conceal these implementation types as a separate concern. Various methods for achieving this are described in the next section.

5.3 Abstraction

In the context of module systems, the term abstraction is used to describe the ability to hide information about the implementation of a module and to protect it against misuse. This is an important feature in practical systems but the use of parameterized signatures described in this paper does not itself provide any way of constructing an abstract datatype.

In fact, we regard this as a distinct advantage because it allows us to treat the issue of abstraction as a separate concern. For example, one possibility is to include support for existential types, perhaps using the approach described in Section 2.1, or the dot-notation [4], or the combination of type inference and existential typing that has been explored by Läufer [12].

However, we have also seen that existential types are not always appropriate. Fortunately, it is also possible to extend the language in a modular fashion with constructs that allow the programmer to provide a name for a datatype but to restrict the scope of its constructors and selectors to a particular collection of bindings. This is essentially what the **abstype** construct used in several different languages achieves, and is closely related to the concept of Skolemization in predicate logic. Note that this does not require any changes to the underlying type theory and is perhaps best dealt with at the level of compilation units rather than the core language.

We believe that both of these approaches are useful in their own right. However, neither coincides exactly with the form of abstract datatypes provided by generativity in SML which falls somewhere between the two extremes of named and existentially quantified abstract datatypes. It seems unlikely that there is a modular extension of our system that provides exactly the same form of abstraction as SML.

6 Conclusion

There are a number of proposals for type-theoretic foundations of modular programming. Some of these systems are very powerful, but require significant and complex extension and modification to account for features that are useful in practice.

Our work shows that the ever-increasing complexity that we have seen in recent work to formalize module systems *can* be avoided and that other, simple, expressive, and viable options are available.

In this paper, we have presented a simple type system that provides:

- Support for higher-order polymorphism.
- Support for structures with polymorphic components.
- A clear separation between static and dynamic semantics.

This leads to a module system in which:

- Structures are first-class values.
- Higher-order modules (i.e., first-class functors) are admitted.
- Polymorphic modules and structures may be defined.
- True separate compilation is possible.
- Parametric polymorphism plays a major role, making the system easier to learn for programmers who are already familiar with the core languages of SML, Haskell or similar languages.

By contrast, none of these is possible with the SML module system.

One of the main topics for future research is to investigate the role of implicit subsumption; that is, the ability to

discard elements from a structure as a result of signature matching in SML. We believe that this can be accomplished using a simple form of subtyping, guided by type annotations, or otherwise by extending the system with a mechanism for controlling the set of bindings that are exported from a structure.

Acknowledgements

Some of the ideas presented in this paper were developed while the author was a member of the Department of Computer Science, Yale University, supported in part by a grant from ARPA, contract number N00014-91-J-4043.

Thanks to Paul Hudak, Sheng Liang, Bob Harper, Colin Taylor and, in particular, Dan Rabin, Xavier Leroy, and Graham Hutton for their valuable comments and suggestions during the development of the ideas presented in this paper. Thanks also to Paul Hudak, Linda Joyce and Chih-Ping Chen for their help in preparing the original submission.

References

- [1] María Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993.
- [2] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.
- [3] Sandip K. Biswas. Higher-order functors with transparent signatures. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 1995.
- [4] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report report 56, DEC SRC, 1990.
- [5] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.
- [6] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.
- [7] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.
- [8] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture*, pages 190–205, New York, 1985. Springer-Verlag. Lecture Notes in Computer Science, 201.
- [9] Mark P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.
- [10] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.
- [11] Stefan Kahrs. First-class polymorphism for ML. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.
- [12] Konstantin Läuffer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.
- [13] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.
- [14] Xavier Leroy. A syntactic theory of type generativity and sharing. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, FL, June 1994.
- [15] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 1995.
- [16] David MacQueen. Using dependent types to express modular structure. In *13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, FL, January 1986.
- [17] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems – ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.
- [18] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*. North Holland, Amsterdam, 1982.
- [19] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.
- [20] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.
- [21] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1988.
- [22] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

- [23] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
- [24] Eugenio Moggi. A category-theoretic account of program modules. In *Summer conference on category theory and computer science*, pages 101–117, New York, 1989. Springer-Verlag. Lecture Notes in Computer Science, 389.
- [25] S.L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, 1987.
- [26] Z. Shao and A. Appel. Smartest recompilation. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993.
- [27] Mads Tofte. Principal signatures for higher-order program modules. In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, January 1992.
- [28] P. Wadler. The essence of functional programming (invited talk). In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–14, Jan 1992.